ISSN 2039-7941
Anno 2016_Numero 360

rapporti tecnici INGV

Parallel performance of the new INGV-PI Laki cluster measured with HPL, HPCG, OpenFOAM and ASHEE

360

Istituto Nazionale di Geofisica e Vulcanologia

REGISTRAZIONE AL TRIBUNALE DI ROMA N.173 | 2014, 23 LUGLIO
© 2014 INGV Istituto Nazionale di Geofisica e Vulcanologia
Rappresentante legale: Carlo DOGLIONI
Sede: Via di Vigna Murata, 605 | Roma

ISSN 2039-7941
Anno **2016**_Numero **360**

rapporti tecnici INGV

PARALLEL PERFORMANCE OF THE NEW INGV-PI LAKI CLUSTER MEASURED WITH HPL, HPCG, OPENFOAM AND ASHEE

Stella V. Paronuzzi Ticco[1,2,3], Luca Nannipieri[1], Matteo Cerminara[1,2], Tomaso Esposti Ongaro[1]

[1]**INGV** (Istituto Nazionale di Geofisica e Vulcanologia, Sezione di Pisa)
[2]**OGS** (Istituto Nazionale di Oceanografia e Geofisica Sperimentale, Trieste)
[3]**MHPC**, the joint SISSA-ICTP master program in HPC, Trieste

360

Index

1. Laki Hardware Architecture

Laki is a High Performance Computing system installed at INGV-Pisa to support the fluid-dynamic and volcanology modelling activity.

Laki is built with a traditional architecture used for high-performance computing clusters as pioneered by the Network of Workstations project [Anderson et al., 1995] and popularized by the Beowulf project [Sterling et al., 2005]. This system is composed of standard high-density servers, a Gigabit Ethernet network and a high-performance InfiniBand interconnection (Figure 1). We have defined the cluster architecture to contain a minimal set of high-density components in an effort to build reliable systems by reducing the component count and by using components with large mean-time-before-failure specifications. Information and access to specific monitoring tools can be found at http://laki.pi.ingv.it; the site can be reached only from inside INGV network, or through an INGV VPN connection.

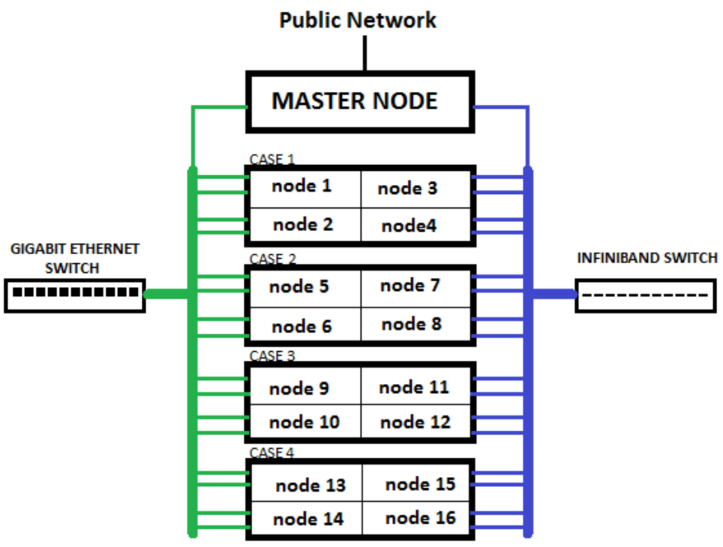


Figure 1. Laki Hardware Architecture.

1.1 Master and Computing Nodes

The cluster is made by 4 chassis containing 4 nodes each and 1 Master Node. Such architecture saves energy and space because a single chassis can manage more efficiently powering and cooling of multiple nodes. All 16 computing nodes are equipped with a dual processor board with the following specifications:

Each node has 2 CPU Intel(R) Xeon(R) CPU E5-2630 v3 (Haswell), with the following specifications:

- 8 cores;
- 2.40 GHz base processor frequency (Turbo Boost at 3.20 GHz);
- 64 bit instruction set;
- 4 memory channels;
- Integrated Memory Controller.

RAM is composed of 8 DDR4 SDRAM banks, each one with 8GB capacity, with clock speed of 2133 MHz, for a total of 64 Gbytes per board. Memory cache is internal to the processor and is distributed as follows:

	L1 cache	L2 cache	L3 cache
Physical ID	77	78	79
Slot	CPU Internal L1	CPU Internal L2	CPU Internal L3
Capacity	512KiB	2MiB	20MiB
Capabilities	internal write-back	internal write-back unified	internal write-back unified

On each computing node, a single hard disk (1 TB SATA 3 7.2K Rpm) is installed, which hosts the Operating System. The master node has a LSI 9361-4i RAID controller which manages 10 hard disks (4 TB SAS 7.2K Rpm) in RAID 5 configuration (for a total volume size of 36 TB, used for users home storage) and 2 hard disks (500 GB SATA 3) in RAID 1 configuration for the operating system. The nominal average dissipated thermal power (TDP) is 85 W per processor and the maximum consumption is 6.5 kW. The Laki cluster measured power consumption is 2.7 kW without load.

1.2 Interconnection: Gigabit Ethernet and Infiniband Switches

The Gigabit Ethernet switch has 24 RJ-45 autosensing 10/100/1000 Mbps ports, with 104 Gb/s routing/switching capacity. All the compute nodes are connected via Gigabit Ethernet to a front-end. Infiniband switch has 18 40/20/10 Gbps auto-negotiation ports, with switching capacity of 2.88Tbps. The Ethernet network is used for the cluster management and monitoring (through SNMP – Simple Network Management Protocol), scheduling (through SSH – Secure Shell) and I/O (through the NFS – Network File System), while the Infiniband network is used for communications (through the MPI – Message Passing Interface).

2. Software Architecture

In November of 2000, the SDSC Grids and Clusters group (www.sdsc.edu) released the first version of the NPACI Rocks cluster toolkit. NPACI Rocks (www.rocksclusters.org) is a complete cluster-aware Linux distribution based upon CentOS (www.centos.org) with additional packages and programmed configuration to automate the deployment of high-performance Linux clusters. The CentOS distribution was chosen because of two key features: 1) the software packaging tool (RPM), and 2) the script-driven software installation tool (Kickstart), that describes a node software stack. By utilizing RPM and Kickstart, a mechanism has been developed that support fully-automated node installation, an important feature when deploying clusters. Although the focus of Rocks is on flexible rapid system configuration (and re-configuration), the steady-state behaviour of Rocks has the look and feel much like any other commodity cluster containing de facto cluster standard services (e.g., Portable Batch System - PBS Torque, Ganglia monitoring and Message Passing Interface – MPI).

2.1 Kickstart and RPM

Kickstart is the CentOS-provided mechanism for automating system installation. Kickstart, together with CentOS software packaging format (RPM), has enabled cluster builders to specify the exact software package and software configuration of a system. This textual description is then used to build a software image on the target platform. Although managing a single Kickstart file can be simpler than managing cluster file images, Kickstart is limited in terms of programmability. The lack of a macro language and a code re-use model potentially requires a unique Kickstart file for every node in a cluster. Rocks solves this problem by generating Kickstart files on-the-fly based on the programmatic target system description, and the site-specific configuration data stored in a MySQL database. Rocks replaces the static file with a script (CGI) to dynamically produce a node-specific Kickstart file. The integration of a node in a cluster requires the node to boot an installation kernel using network boot (PXE) or a physical boot media (e.g., CDROM or hard disk). This installation kernel requests a Kickstart file over HTTP, and executes this file installing the appropriate software packages and applying software configuration. The process of installing software on any system always contains two components: the installation of software packages and the configuration of software packages. Often the configuration of a package is such that it simply accepts the defaults, but sometimes a different configuration from the default is required. The traditional single desktop approach to this process is to install software and then, manually, to configure it according to specific requirements. A common extension of this process to the cluster deployment, is to manually configure a single node and then propagate the resulting system image to all the nodes in the cluster. This works well with homogeneous hardware. Rocks treats software installation and software configuration as separate components of a single process. This means that manual configuration required to build a file image is instead automated. Installation of software packages is done in the form of package installs according to the functional role of a single cluster node. This is also true for the software configuration. Once both the software packages and software configuration are installed on a machine, we refer to the machine as an appliance. Rocks clusters contain Front-end, Compute, and NFS appliances (Network File System). A simple object-oriented framework (expressed in XML) is used to allow cluster architects to define new appliance types and take full advantage of code reuse for software installation and configuration.

2.2 The module environment

The module package (https://modules.sourceforge.net) is a useful package with which the user can, via the 'module' command, use additional software not included in Rocks release. On the Laki cluster, instead of the default OpenMPI (1.6.2) installed through Rocks, a custom OpenMPI (1.6.5) has been rebuilt with support for Torque and Intel Infiniband switch, installed under /share/app/ path and inserted in the module configuration file. The directory /share/app/ is on master node and shared with all nodes; the users can load the OpenMPI 1.6.5 environment with the 'module load openmpi-1.6.5' command. Several other environment definitions can be loaded via the module command, and a full list of these packages can be obtained issuing a 'module avail' command.

2.3 PBS Torque

As for every high computing facility, we have installed a batch system, i.e. a computer application for controlling unattended background program execution of jobs. This is commonly called program batch scheduling (PBS), as execution of non-interactive jobs is often called batch processing, while the data structure of jobs to run is known as the job queue. We went for the TORQUE (Terascale Open-source Resource and QUEue Manager, https://wikipedia.org/TORQUE) resource manager, as it provides enhancements over standard OpenPBS like greater fault tolerance, and higher scalability [7, 8]. Each user can submit his simulation using simple bash scripts that specify parameters requested by the batch system, as PBS job name, number of requested cores or desired reserved time. In the present configuration, no parameters are taken into account to assign a job priority. All the jobs are simply executed as soon as there are enough available resources on the machine, and run on until the requested time expires. As soon as the usage of the machine would increase, or the number of users grow, it will be possible that parameters as the mean time of usage of machines, per research group, or the level of optimization of the jobs submitted will be taken into account to assign a job priority.

2.4 Ganglia Monitoring Tool

This cluster is equipped with the Ganglia tool, a distributed monitoring system for high-performance computing systems such as clusters and Grids (https://wikipedia.org/Ganglia).

This tool can be accessed via web (http://laki.pi.ingv.it/wordpress) and allows the user to monitor the usage of the machines, using carefully engineered data structures and algorithms to achieve very low per-node overheads and high concurrency. It reports substantially what a user manually would see connecting to a node and issuing a "top" command, but in a more compact and understandable way, through graphs that report also historical statistics (such as CPU load averages or network utilization) for all machines that are being monitored.

3. The HPL Linpack Benchmark

To test the hardware/software installation and the cluster performance, we have run the HPL Linpack benchmark, a portable implementation of HPLinpack used to provide data for the Top500 list (https://www.top500.org/lists). HPL consists of an algorithm for solving a general dense matrix problem $Ax = b$ in 64-bit floating point arithmetic. The general idea is to decompose the matrix A into the product of simpler, well-structured matrices, which can be easily manipulated to solve the original problem. This goal is achieved using an algorithm based on LU decomposition with partial pivoting; the matrix type is real and dense, with matrix elements distributed randomly in the range $[-1, 1]$. Once the system has been solved, the input matrix and the right-hand side are regenerated and the residuals are computed [Dongarra et al., 2010]. The package uses 64-bit floating-point arithmetic and portable routines for linear algebra operations and message passing. The latter is implemented through OpenMPI, while for the linear algebra several possibilities exist. After testing the standard BLAS libraries, we decided to go for the open-source optimized OpenBLAS implementation of a Basic Linear Algebra.

CPU	Intel(R) Xeon(R) CPU E5-2630 v3
Speed	2.40 GHz
OS	CentOS release 6.6
OpenMPI	openmpi-1.6.5/openmpi-1.6.2
C compiler	mpicc
C Flags	-fomit-frame-pointer -O3 -funroll-loops
BLAS	openblas 0.2.18
Link layer	InfiniBand QDR IB (40Gb/s)

Table 1. Specification of Hardware and Software used for benchmarks.

3.1 HPL Setup

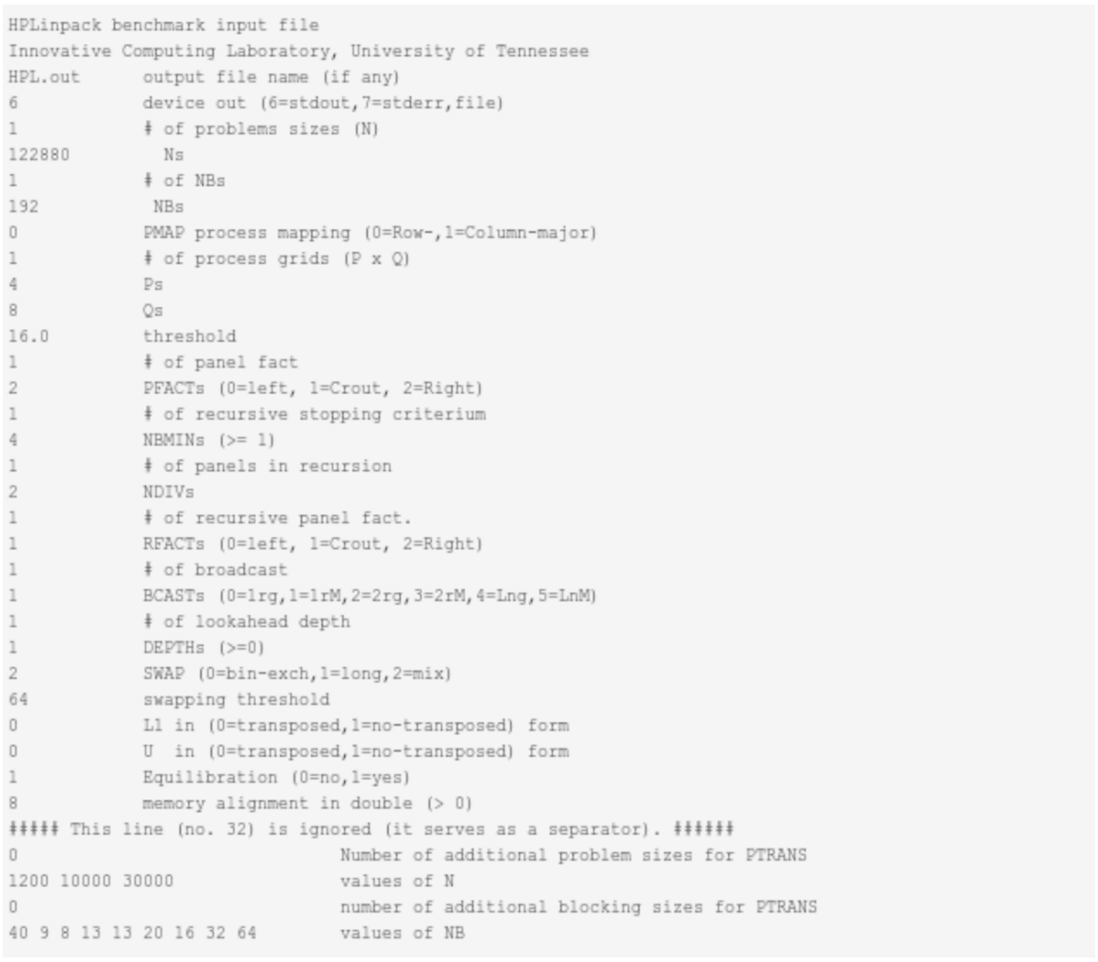
Once the benchmark is correctly compiled and all its dependencies are satisfied (i.e. a proper BLAS library has been installed), the executable is built. To run the benchmark one must provide an input file that specifies several options available for HPL. One has to tune this input file in order to obtain the best possible performance out of the cluster: this is not a trivial problem at all, as a large number of options are available and some of them imply a rather deep understanding of the implemented algorithm, more than a basic knowledge of the involved numerical methods.

We used as a start the input file generated at www.advancedclustering.org, then tuned manually some of the parameters. Figure 2 reports a typical HPL input file. The ones relevant for this report are:

- *N*: specifies the size of the matrix to decompose.
- *NB*: specifies the block size[1].
- *P*, *Q*: specify the number of process rows (*P*) and columns (*Q*) of the execution grid, which is equal to the number of available cores.

The last two parameters specify also the number of cores used by the run: it must be less than or equal to $P*Q$. In particular HPL prefers "square" or slightly flat process grids. Unless one is using a very small process grid, is better to stay away from the 1-by-Q and P-by-1 process grids. The remaining lines allow specifying algorithmic features (please visit www.netlib.org/HPL/tuning for details).

[1] HPL uses the block size NB for the data distribution as well as for the computational granularity. From a data distribution point of view, the smallest NB, the better the load balance. From a computation point of view, a too small value of NB may limit the computational performance by a large factor because almost no data reuse will occur in the highest level of the memory hierarchy. The number of messages will also increase. Efficient matrix-multiply routines are often internally blocked. Small multiples of this blocking factor are likely to be good block sizes for HPL.



```
HPLinpack benchmark input file
Innovative Computing Laboratory, University of Tennessee
HPL.out      output file name (if any)
6            device out (6=stdout,7=stderr,file)
1            # of problems sizes (N)
122880         Ns
1            # of NBs
192           NBs
0            PMAP process mapping (0=Row-,1=Column-major)
1            # of process grids (P x Q)
4            Ps
8            Qs
16.0         threshold
1            # of panel fact
2            PFACTs (0=left, 1=Crout, 2=Right)
1            # of recursive stopping criterium
4            NBMINs (>= 1)
1            # of panels in recursion
2            NDIVs
1            # of recursive panel fact.
1            RFACTs (0=left, 1=Crout, 2=Right)
1            # of broadcast
1            BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)
1            # of lookahead depth
1            DEPTHs (>=0)
2            SWAP (0=bin-exch,1=long,2=mix)
64           swapping threshold
0            L1 in (0=transposed,1=no-transposed) form
0            U  in (0=transposed,1=no-transposed) form
1            Equilibration (0=no,1=yes)
8            memory alignment in double (> 0)
##### This line (no. 32) is ignored (it serves as a separator). ######
0                              Number of additional problem sizes for PTRANS
1200 10000 30000               values of N
0                              number of additional blocking sizes for PTRANS
40 9 8 13 13 20 16 32 64       values of NB
```

Figure 2. Example of HPL Linpack input file.

3.2 HPL Results

In Table 2 we report the results obtained running HPL on 256 cores, using different problem sizes and different run options. The theoretical peak performance is defined as:

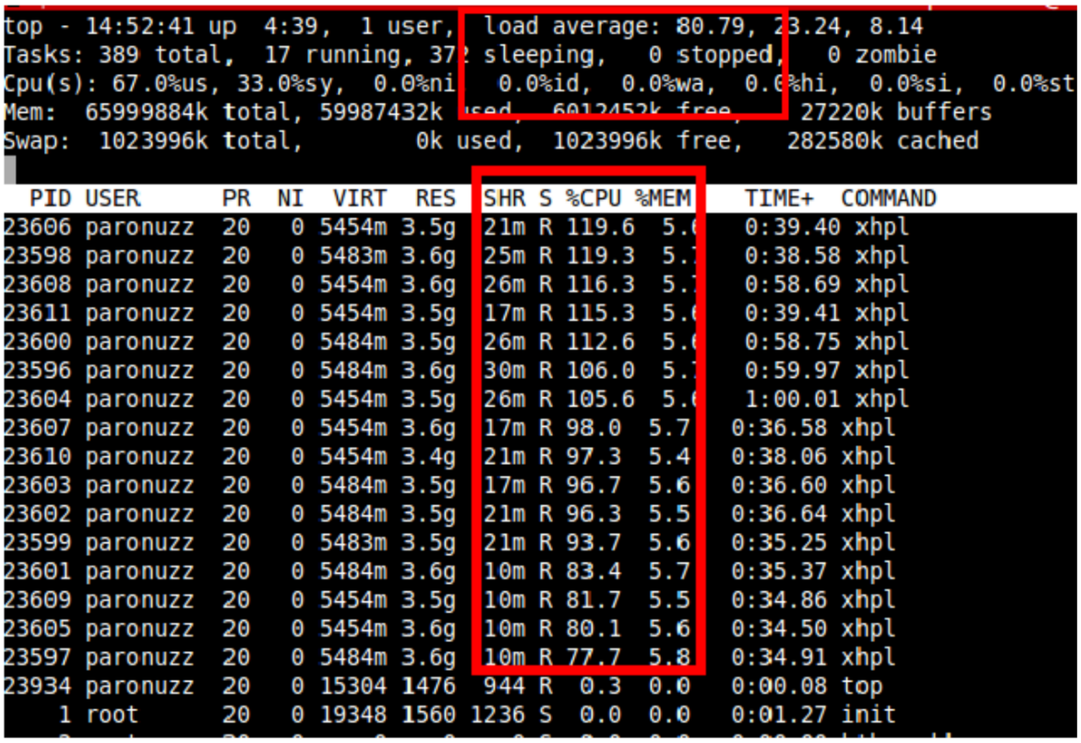
$$\text{FLOPS} = \#\ \text{cores} \times \text{clock} \times \#\ (\text{FLOP/clock})$$

The theoretical peak has to be regarded as an asymptotic limit for the performance. The third factor (the number of floating point operations per clock cycle) is indeed associated to vectorization: nominally, for the E2630v3 CPU, Intel claims 16 operations per clock cycle, which is nonetheless very difficult to achieve. Several factors contribute to lowering this value, depending on the specific application: different algorithms are able to exploit vectorization at different levels, depending on implementation and also on compiling options. In the present case and for this specific benchmark, we obtained a value of about 8 FLOP/(clock cycle).

N	NB	TFLOPS	Ethernet	Infiniband	Blas	OpenBlas
327552	192	2.62	✓			✓
327552	192	2.22		✓	✓	
	192	4.47		✓		✓
347520	128	4.39		✓		✓
	192	4.50		✓		✓
	256	4.46		✓		✓
Theoretical Peak		9.83				

Table 2. Performance (in TeraFLOPS) of HPL Linpack on 256 Laki cores.

The reported results are obtained running with the '-bind-to-core' MPI option (Figure 3 and Figure 4). This is really important as it prevents the OS from moving the processes from core to core in an attempt to optimize channel memory usage. This is useful if the machine load is around 50%, but when the workload is at a peak this creates an amount of requests that is impossible to satisfy, leading to a situation in which a performance degradation on the whole node is likely. Finally, although many tutorials suggest running HPL without the use of a job scheduler, our results demonstrated that running HPL through PBS entails no significant overhead. We have then increased N, trying also different block sizes, obtaining the best estimate of the Laki peak computational power: 4.495 × 10^3 GFLOPS. It is not possible to run bigger problems, as they would saturate the RAM capacity, causing the OS to kill the job.



```
top - 14:52:41 up  4:39,  1 user,  load average: 80.79, 23.24, 8.14
Tasks: 389 total,  17 running, 372 sleeping,   0 stopped,   0 zombie
Cpu(s): 67.0%us, 33.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:  65999884k total, 59987432k used,  6012452k free,    27220k buffers
Swap:  1023996k total,        0k used,  1023996k free,   282580k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
23606 paronuzz  20   0 5454m 3.5g  21m R 119.6  5.6   0:39.40 xhpl
23598 paronuzz  20   0 5483m 3.6g  25m R 119.3  5.7   0:38.58 xhpl
23608 paronuzz  20   0 5454m 3.6g  26m R 116.3  5.7   0:58.69 xhpl
23611 paronuzz  20   0 5454m 3.5g  17m R 115.3  5.6   0:39.41 xhpl
23600 paronuzz  20   0 5484m 3.5g  26m R 112.6  5.6   0:58.75 xhpl
23596 paronuzz  20   0 5484m 3.6g  30m R 106.0  5.7   0:59.97 xhpl
23604 paronuzz  20   0 5454m 3.5g  26m R 105.6  5.6   1:00.01 xhpl
23607 paronuzz  20   0 5454m 3.6g  17m R 98.0  5.7   0:36.58 xhpl
23610 paronuzz  20   0 5454m 3.4g  21m R 97.3  5.4   0:38.06 xhpl
23603 paronuzz  20   0 5484m 3.5g  17m R 96.7  5.6   0:36.60 xhpl
23602 paronuzz  20   0 5484m 3.5g  21m R 96.3  5.5   0:36.64 xhpl
23599 paronuzz  20   0 5483m 3.5g  21m R 93.7  5.6   0:35.25 xhpl
23601 paronuzz  20   0 5484m 3.6g  10m R 83.4  5.7   0:35.37 xhpl
23609 paronuzz  20   0 5454m 3.5g  10m R 81.7  5.5   0:34.86 xhpl
23605 paronuzz  20   0 5454m 3.6g  10m R 80.1  5.6   0:34.50 xhpl
23597 paronuzz  20   0 5484m 3.6g  10m R 77.7  5.8   0:34.91 xhpl
23934 paronuzz  20   0 15304 1476  944 R  0.3  0.0   0:00.08 top
    1 root      20   0 19348 1560 1236 S  0.0  0.0   0:01.27 init
```

Figure 3. Screenshot of the top command. Without the –bind-to-core MPI option, the value of the average cluster load blows up and the FPUs appears to be overloaded (load exceeds 100%) as a result of the overhead introduced by the OS scheduling mechanism.

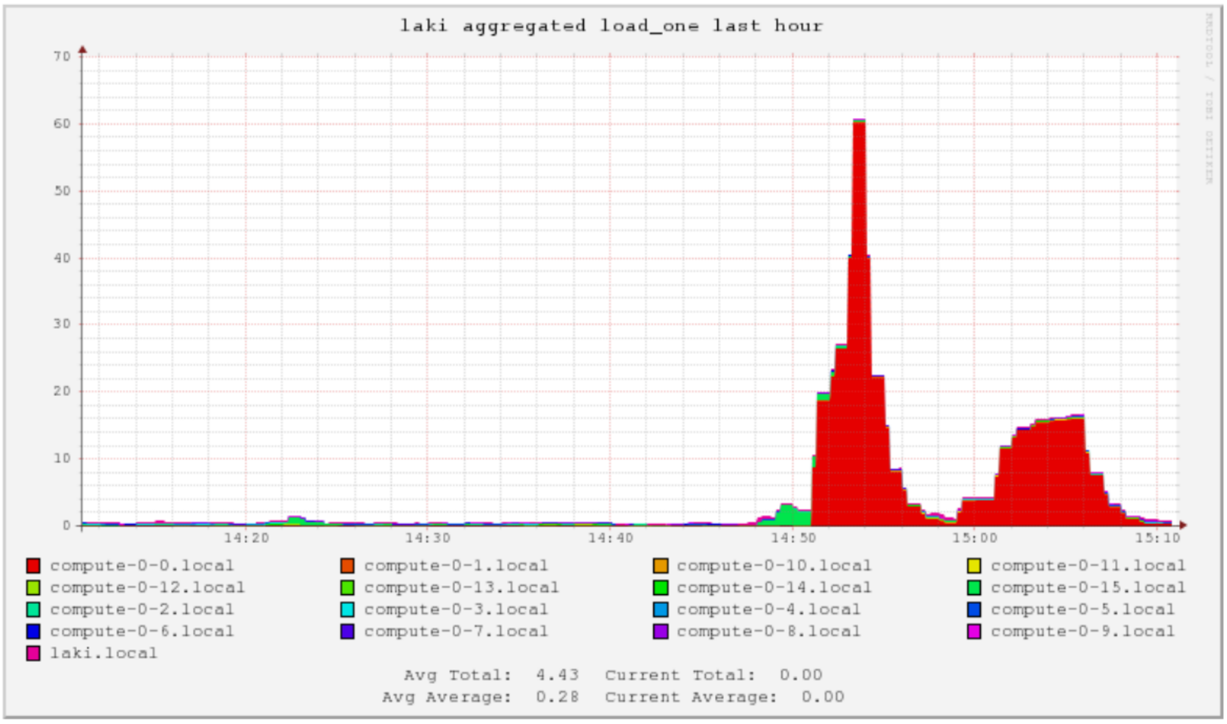


Figure 4. Screenshot of the GANGLIA monitoring tool. The two peaks refer to the same test run without and with the –bind-to core MPI option. Without core binding, the maximum aggregated load is about 4.5 times larger.

3.3 HPL Scaling Performance

Figure 5 represents the performance in GFLOPS, on 256 cores, as a function of the matrix size *N*. The performance clearly deteriorates for both too small and too big problems; this ensures that we have measured the actual peak performance, and not only a local maximum.

Once the peak performance has been measured we studied the scaling properties of the HPL benchmark by keeping the load per processing element constant. For this purpose we used the input file that gave us the peak performance, but with different number of cores, reducing accordingly the size of the problem and adapting the grid. What we obtained is an almost linear scaling (Figure 6): this was quite expected, as the HPL benchmark has been written to measure computing efficiency and not to stress communication. Anyway as pointed out in Figure 9, some problems in the intra-node scaling are already evident, producing a decrease of parallel efficiency to about 0.75 at N=8. This is also one of the main reasons why we cannot stop here our study: real-world application use communication, and do it quite intensively. We can say that the numbers obtained so far are satisfactory, but are not at all exhaustive to give an idea of the performance to be expected during any real use-case.

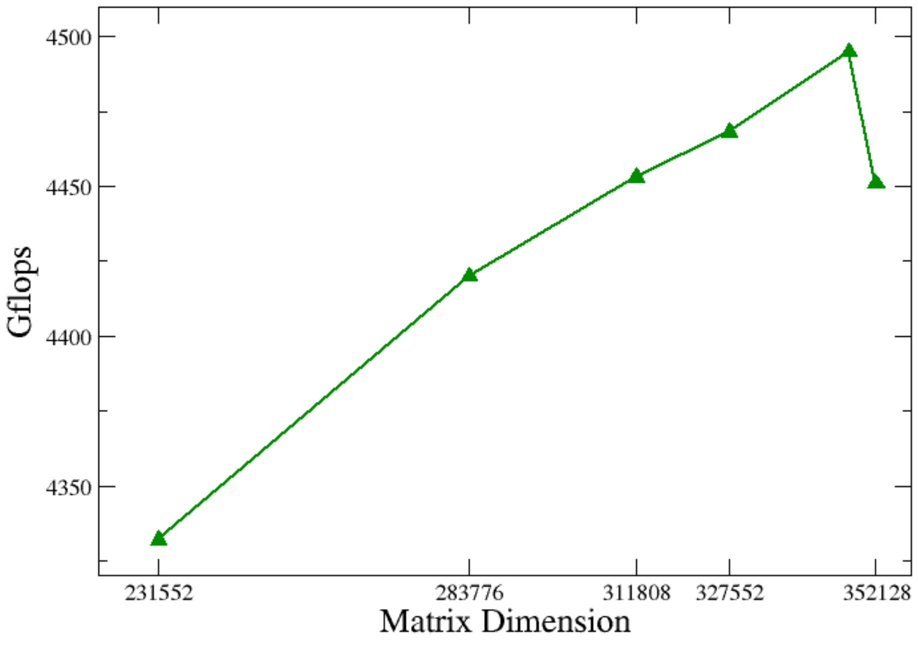


Figure 5. HPL Performance with changing matrix dimension. The curve is monotonically approaching the maximum value, and then decreasing highlighting a saturation of available RAM.

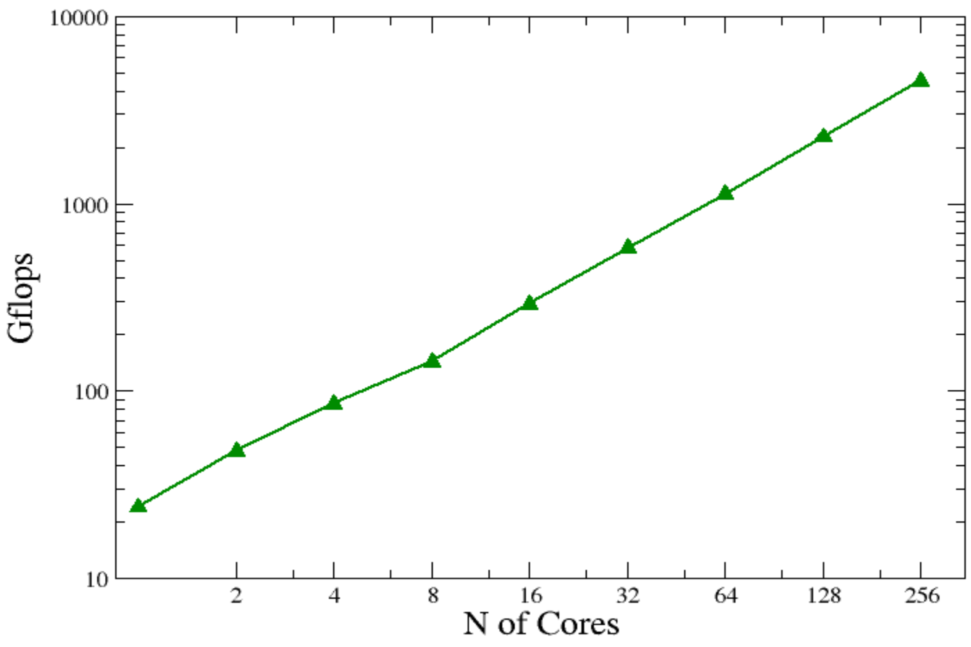


Figure 6. Linear scaling of HPL in log-log scale. Results obtained with the same input file as the optimal benchmark with decreasing matrix size.

4. The HPCG Benchmark

HPL is the most widely recognized and discussed metric for ranking high performance computing systems. However, HPL is increasingly unreliable as a true measure of system performance for a growing collection of important science and engineering applications. The problem is that the dominant calculations in this algorithm are dense matrix-matrix multiplication and related kernels: with proper organization of the computation, data access is predominantly unit stride and is mostly hidden by concurrently performing computation on previously retrieved data. This kind of algorithm strongly favours computers with very high floating-point computation rates and adequate streaming memory systems, but while these types of patterns are commonly found in real applications, additional computations and access patterns are also very common. HPL as a metric is incapable of measuring these types of patterns, but the system ability to effectively address this second kind of patterns is an important indicator of system balance: for this reason in 2013 J. Dongarra and P. Luszczek proposed a new benchmark [Dongarra et al., 2013], called HPCG (High Performance Conjugate Gradient Benchmark). This new application has been regarded as a new metric for high performance computing for various reasons:

1. Provides coverage of the major communication and computational patterns.
2. Represents a minimal collection of the major patterns.
3. Rewards investment in high-performance of collectives.
4. Rewards investment in local memory system performance.

4.1 HPCG Setup

This benchmark generates a symmetric positive-definite matrix *A* and a corresponding right-hand side *b*, plus and initial guess for *x*. In a second time, it sets up data structures for the local symmetric Gauss-Seidel preconditioner, compute pre-conditions, post-conditions and invariants that will aid in the detection of anomalies during the iteration phases. Overall, *m* iterations are performed *n* times, using the same initial guess each time, where *m* and *n* are sufficiently large to test system uptime. By doing this we can compare the numerical results for “correctness” at the end of each *m*-iteration phase. Linear system size is a parameter that can be chosen via an input file: to give relevant results the size has to be large enough so that data arrays accessed in the iteration loop do not fit in the cache of the processor, a condition that would be unrealistic in a real application setting. At the end of every run a log file is produced for reporting information on performance: the HPCG rating is a weighted GFLOP/s value that is composed of the operations performed in the PCG iteration phase over the time taken. The overhead time of problem construction and any modifications to improve performance are divided by 500 iterations (the amortization weight) and added to the runtime. HPCG can be run in just a few minutes from start to finish. However, standard runs must be at least 1800 seconds (30 minutes). Moreover, as stated in the HPCG documentation, the problem size should be large enough to occupy a significant fraction of “main memory”, at least 1/4 of the total.

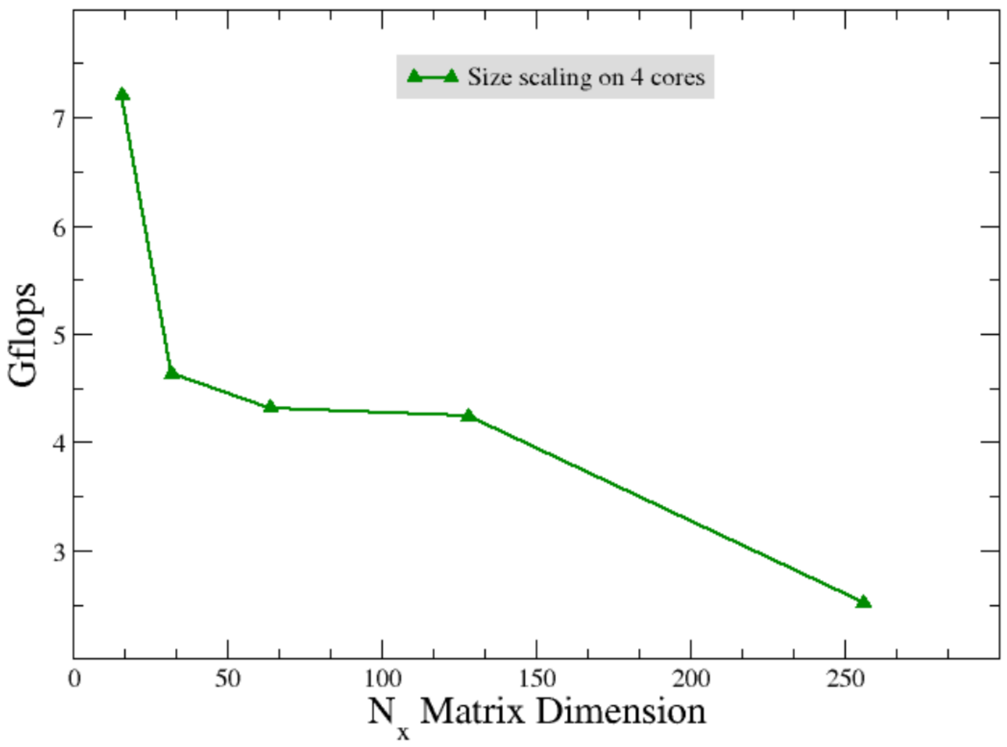


Figure 7. Performance of the parallel HPCG Benchmark with different matrix dimensions.

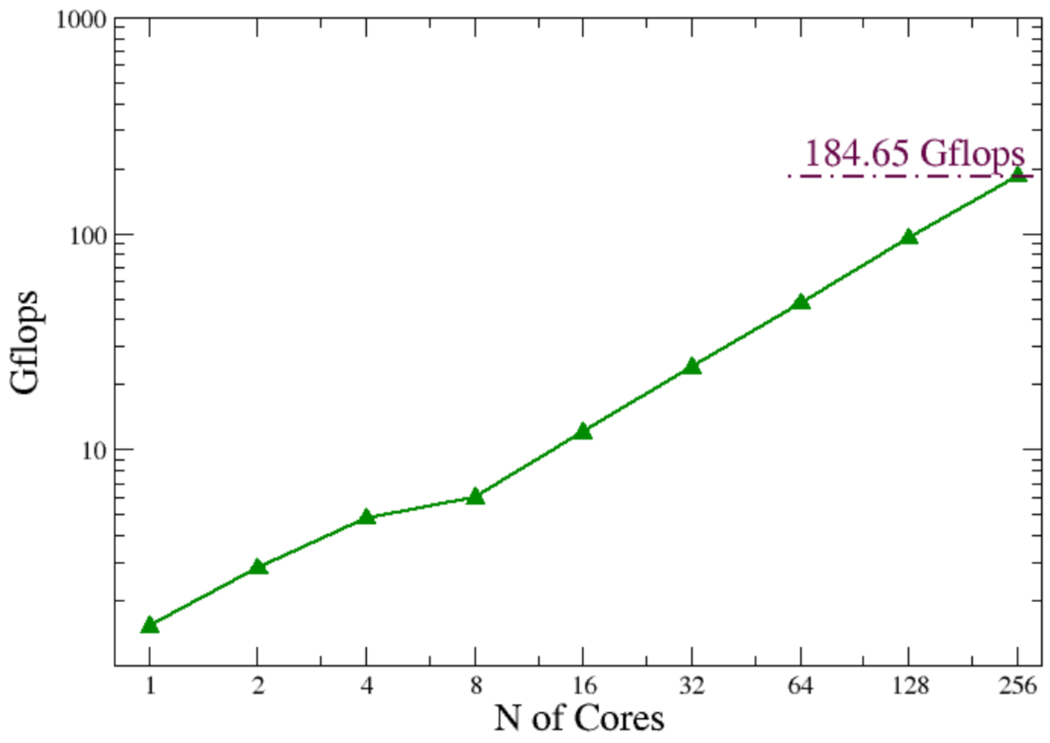


Figure 8. Scaling properties of HPCG in log-log scale, keeping constant the load per processing element (Nx=136). The scaling is linear above N=16.

4.2 HPCG Results

First of all we tuned the problem dimension to obtain the best performance out of Laki cluster (Figure 7): opposite to HPL, increasing the size does not increase performance. Apart for $N_x = 16$, for which the problem fits L3 cache, the performances are pretty much stable, slightly decreasing with increasing matrix size. We chose as the best candidate $N_x = 136$, a value for which 1/4 of the machine memory is already fully occupied.

Once the optimal size is decided we measure the peak performance: we obtain 184.65 GFLOPS (Figure 8), which compared to HPL 4.49 TFLOPS seems a very small performance. However, this was an expected result: it is a well known fact that systems vendor ends up making some either/or decisions about architecture versus tailoring systems to meet HPL application demands. For Laki cluster, the HPL over HPCG ratio is 4.11%. This is a satisfactory result: usually HPCG benchmark gives peak performance orders of magnitude smaller than HPL benchmark, the ratio between HPL and HPCG never exceeding 10% (see Appendix A for more detailed numbers).

Focusing on scaling, the linear behaviour for the number of cores greater than 8 is clear. For a number of cores ≤ 8, i.e. in the intra-node part of the curve, the trend is less than linear and not at all satisfying. This was also observed on the HPL Linpack application, but less visible, due to the different communication pattern implemented in the benchmarks; however, this is clearly depicted in Figure 9, where efficiency is shown as the ratio $(T_1 / n * T_n)$, with T_1 and T_n execution time on 1 and n cores, respectively. As already said at the beginning of this section, the HPCG benchmark implements more common and realistic load scenarios for the machine: this scaling behaviour will be in fact retrieved for real case applications as OpenFOAM and ASHEE (see Sec. 5.2, 5.4).

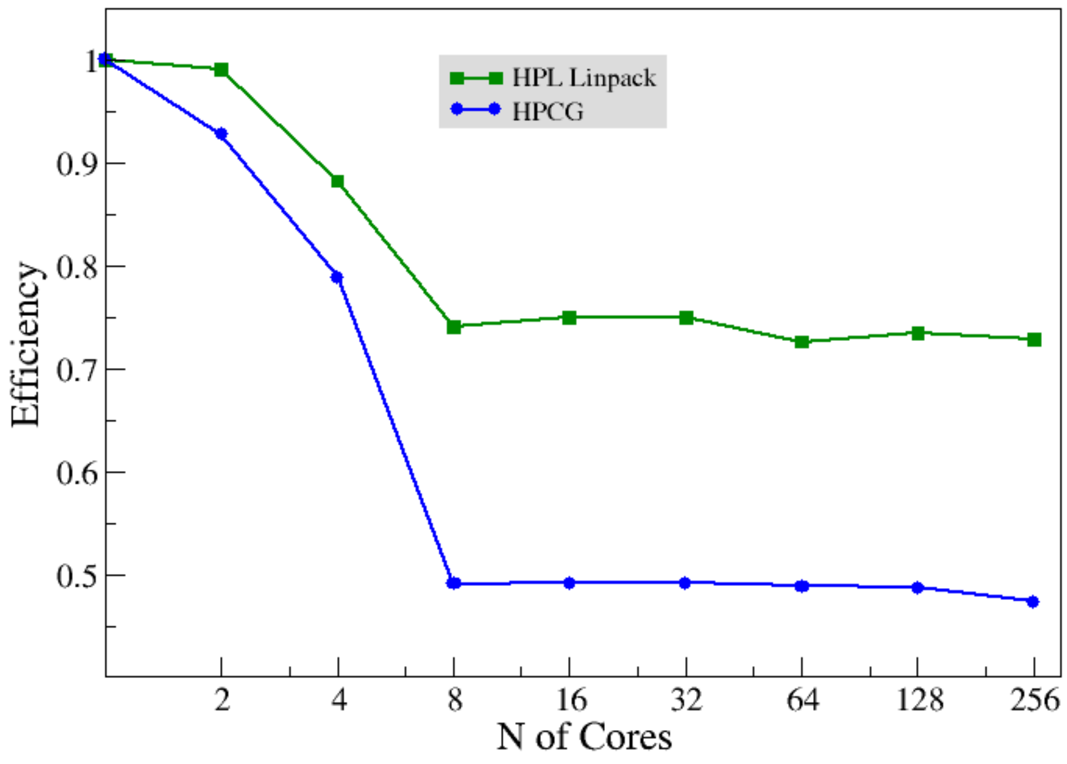


Figure 9. Comparison of the Efficiency of HPL and HPCG on the Laki cluster. The behaviour is non-linear in both cases in the intra-node part of the curve. Such a trend is clearer for HPCG because of the more intense communication pattern.

5. The OpenFOAM Benchmark

A large number of simulations will be performed on the Laki cluster with OpenFOAM® (www.openfoam.org). For this reason, we have run a simplified use case, to try to identify some "best practices" for OpenFOAM simulations on Laki. Before starting with the actual benchmarks we studied how is OpenFOAM performing with different basic OpenMPI options.

5.1 An Embarrassingly Parallel Application

A plethora of options exists for MPI commands and, in particular, for the 'mpirun' command. To get a basic understanding of which option is preferable, we built some very simple OpenFOAM applications that perform basic operations, namely:

1. Copy a constant vector of size 10^6 to another vector (no thread inter-core communication).
2. Allocate a vector of 10^6 random numbers (no communication among processors).
3. Allocate and copy a vector of 10^6 random numbers to another vector (no communication).
4. Allocate, copy and interpolate a vector of 10^6 on the faces (with communication).

These are the various MPI options (www.open-mpi.org) used:

- bind-to-none: default MPI behaviour; allow the OS to manage the binding of the process to the cores, in attempt to balance loads.
- bind-to-socket: bind each MPI process to a processor socket. The OS is still able to move processes between different cores.
- bind-to-core: bind each MPI process to a core, preventing the OS to move it from core to core during the run.
- loadbalance: tries to spread processes out fairly among the nodes.

We performed a test with this trivial application to measure the parallel performance of the more basic part of OpenFOAM: finite volume vector treatment, with and without inter-core thread communication. For each combination of OpenMPI options, we repeated each test 100 times in order to compute the average and standard deviation. These tests were made reserving 64 (or 16) cores for each simulation (through PBS parameters) and then specifying with the -np option the number of cores to use for the actual simulation.

Figures 10-14 report the code efficiency of the different procedures with the different mpirun options. Numbers in legend of Fig. 13 and 14 report the efficiency (with their standard deviation) of each test performed using all the reserved cores. As a general statement, the mpirun options can bring remarkable changes in performance, even for very simple applications not involving any MPI communication. This is likely due to interaction of the application with the operating system and to bottlenecks associated with memory access.

For all the tests a rather high variability in performance is also evident, as reflected by the large standard deviations. It is clearly important that the performance of a code is stable, so one of the objectives is to reduce the standard deviation. In order to address this issue, we explore the best bind- option to be used. From Figs. 13 and 14 (see the legend) we conclude that the -bind-to-core option is the one resulting in the lowest performance variability.

Analysing the graphs (Figs. 10-14), it seems that the -loadbalance option gives the best performance results (showing sometimes super-linearity). However, these graphs are a little misleading. The reason is that this option is useful if one can access a number of cores larger than the number of used MPI processes: in this case, mpirun will try to manage jobs among the nodes to optimize load values. But this also implies that many cores are no longer accessible by other applications. When the machine nodes are saturated (last point of the curves in Figs. 10-14 this option has little impact on performance. Moreover, we observe that the advantages of the –loadbalance option disappear when using a –bind option, since the OS can no longer move the threads. We conclude that the –loadbalance option is useless for our HPC applications.

Communication is not the main issue causing the deterioration of performance up to 64 cores (this is also true up to 256 cores, see next sections). Indeed, solving a problem with or without inter-thread communication does not influence much the performance. They are more likely influenced by the memory access. Comparing Fig. 13 and 14, we notice that the efficiency decreases to ~ 0.6 between 1 and 8 threads, because of the saturation of the 4 memory channels of each 8-cores CPU. Then, increasing the number of cores used, the efficiency remains optimal, meaning that once half of the node is full, MPI is working in ideal conditions.

We expect to find this intra-node behaviour in all the OpenFOAM applications, since it is present even in the simplest finite volume application possible.

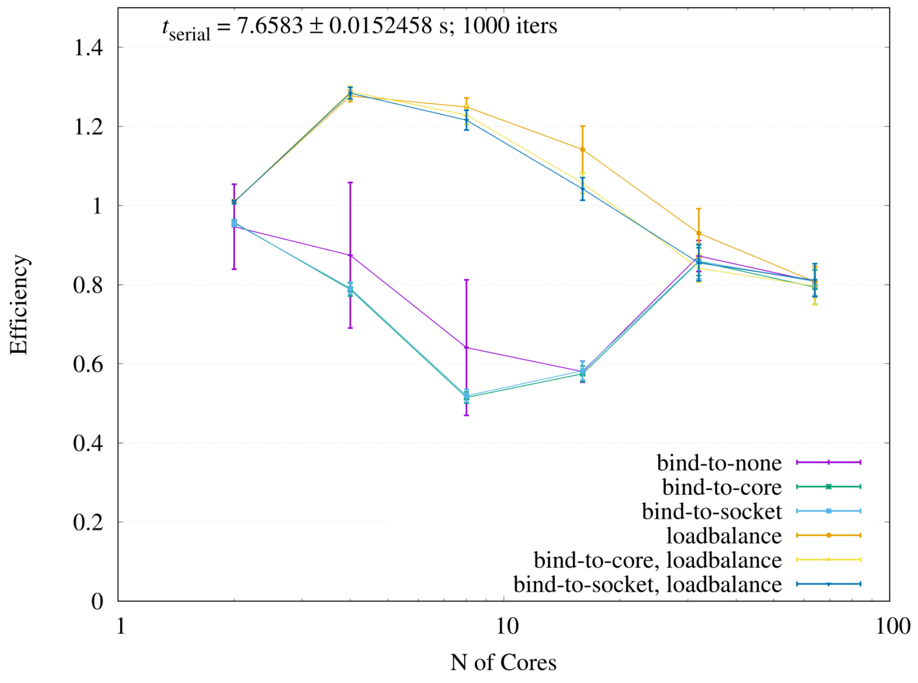


Figure 10. Efficiency of copying a constant vector to another, using 1 to 64 cores, always reserving 64 cores. Different combinations of MPI options are used. The error bars are evaluated computing the standard deviation from 100 identical runs.

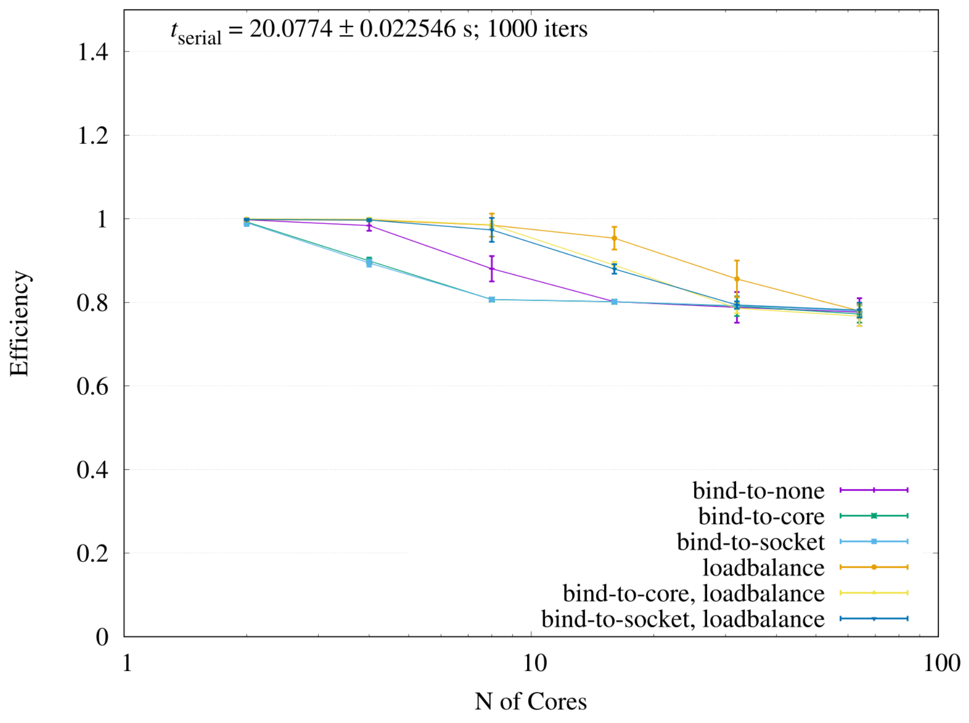


Figure 11. As in Figure 10, but for the allocation of a random vector.

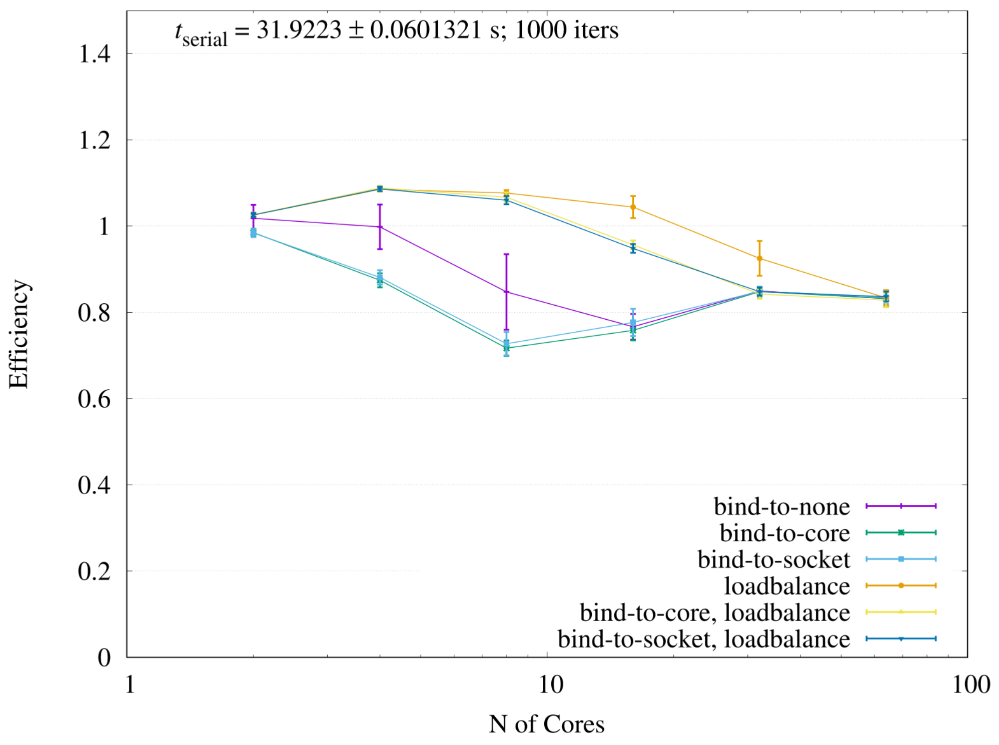


Figure 12. As in Figure 10, but for the allocation and copy of a random vector.

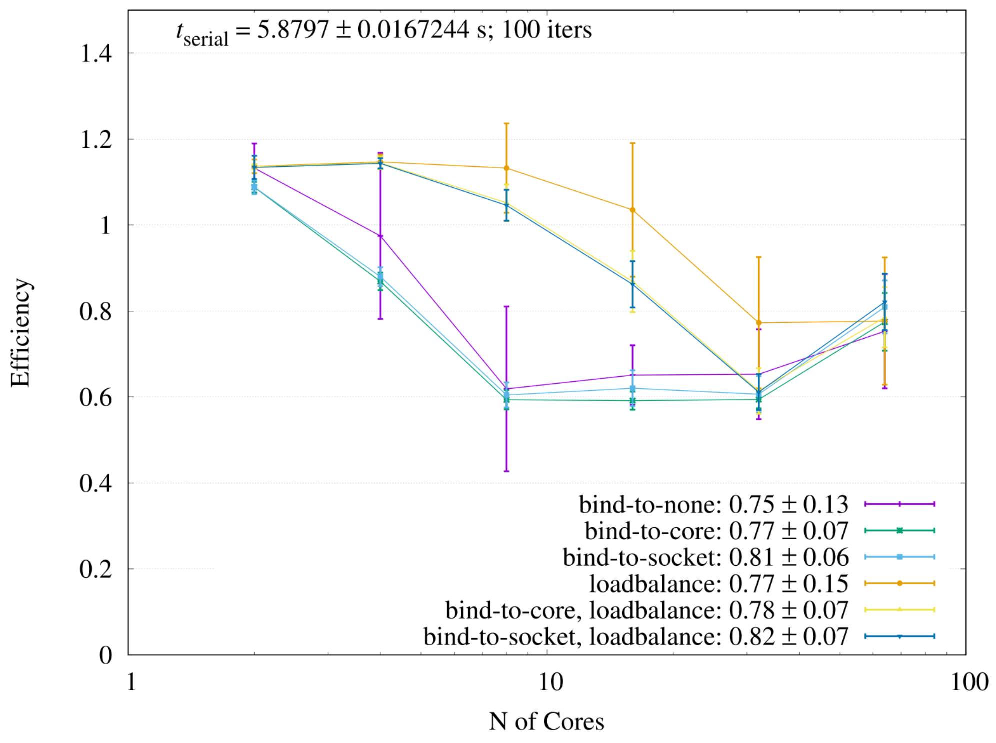


Figure 13. As in Figure 10, but for the allocation, copy and interpolation of a random vector.

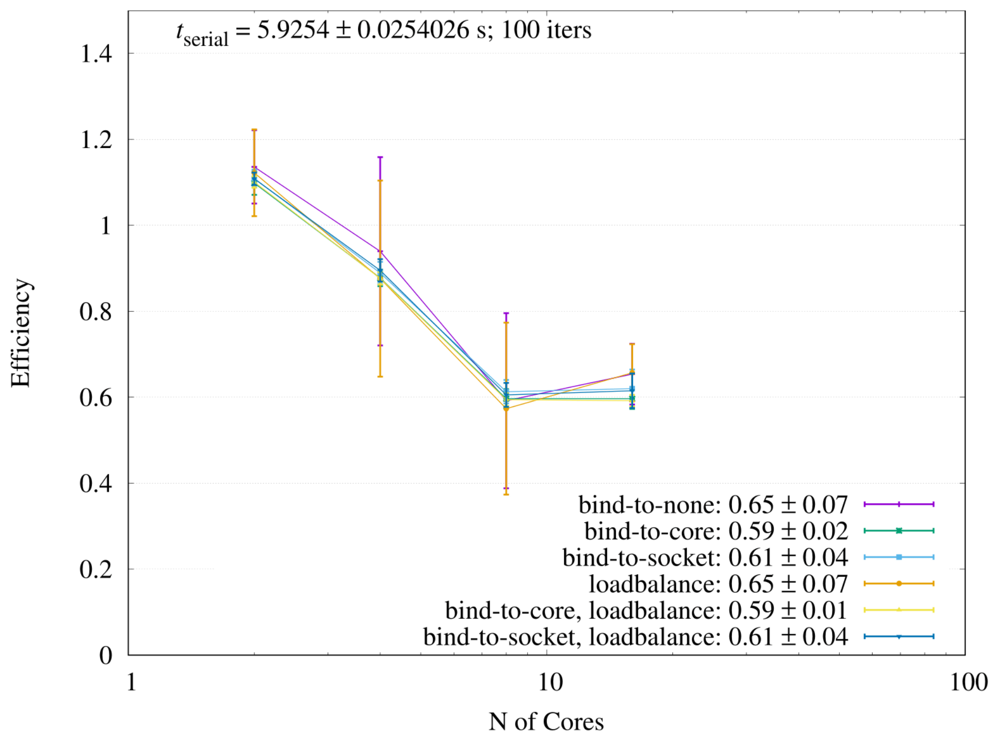


Figure 14. Same of Figure 13, but with 16 cores reserved.

5.2 rhoPimpleFoam Case

To test the cluster performance on a more realistic test case, we run the 'rhoPimpleFoam' solver for compressible fluids (as implemented in the standard distribution of OpenFOAM 3.0), as it is similar to ASHEE, a code written and used at INGV to simulate volcanic ash plumes [Cerminara et al., 2016]. We have set up a forced plume test case, in which a high-temperature jet of fluid flows through an inlet, entering into a box.

The solver does not account for gravity, so the jet simply goes toward the top of the box, generating a turbulent flow of compressible fluid with barotropic boundary conditions [Cerminara et al., 2014] (Fig. 15). To measure the performance, we have used the total elapsed time of a simulation with a fixed number of time-steps. We run two different sets of simulations, with different mesh sizes (Table 3) and tested different OpenFOAM options combined with OpenMPI directives. In addition, the 'renumberMesh' utility of OpenFOAM has been tested. This is executed by the command 'mpirun -n N renumberMesh -overwrite –parallel' with *N* equal to the number of cores. This is useful when running on large numbers of cores to renumber the cell list in order to reduce the memory bandwidth (https://openfoamwiki.net/RenumberMesh).

N of cells	tot	inlet	z	x & y
low res	1.65e+05	4	22	72
high res	7.11e+06	14	77	252

Table 3. Two different mesh resolutions have been tested for the OpenFOAM benchmark.

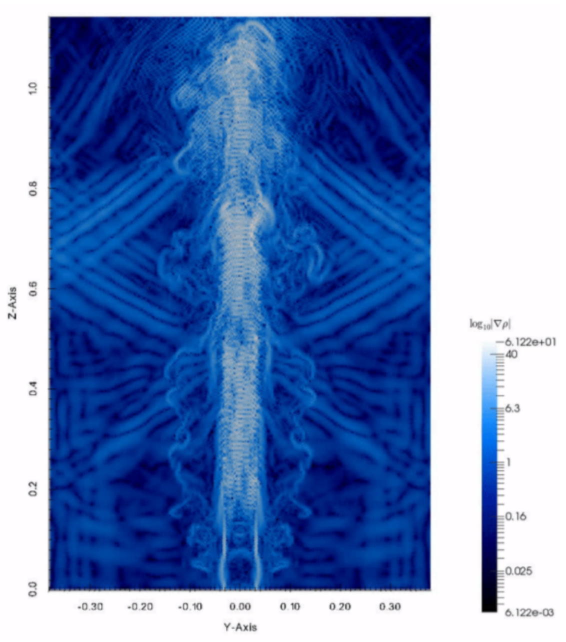


Figure 15. Forced plume simulation with rhoPimpleFoam. Colour scale indicates the density gradient, highlighting shock waves.

We started studying the best MPI + OpenFOAM setup to use when running on multi-core machines. Fig. 16 highlights again the importance of the -bind-to-core MPI option: for a number of cores larger than the number of memory channels per node (8 on Laki cluster) the OS default behaviour -bind-to-none is clearly deteriorating the scaling properties of the problem. The overall scaling efficiency with -bind-to-core option is reported in Figure 17 for two different problem sizes.

To try to explain the intra-node behaviour of the code, we have introduced timing instructions in the solver to report the cumulative execution time of the main routines, for each time step. They have been subdivided in four main categories: matrix assemblage; vector assemblage; matrix inversion; LES (Large

Eddies Simulation) routines. These four contribute to almost 100% of the total execution time per time step. In Figure 18 we report the results obtained on 4 cores for the high-resolution mesh: in the first part of the simulation the scaling efficiency is near 100%. Then simulation time starts to increase at a random time-steps. This increase cannot be ascribed to a single routine, but it is instead common to all the processes. This tests have been repeated more than once, to see if some patterns can be highlighted, but the results have been discouraging: these jumps appear always, but at different time-steps, and also their height differs from run to run. A possible reason for this behaviour is again the memory access. The “steps” in the execution time per time step are possibly related to the progressive saturation of the various memory levels. This would explain why they are almost absent when we reduce the memory request by increasing the number of cores (Figure 19). However, to thoroughly analyse this problem, specific profiling/monitoring tools should be applied in the future.

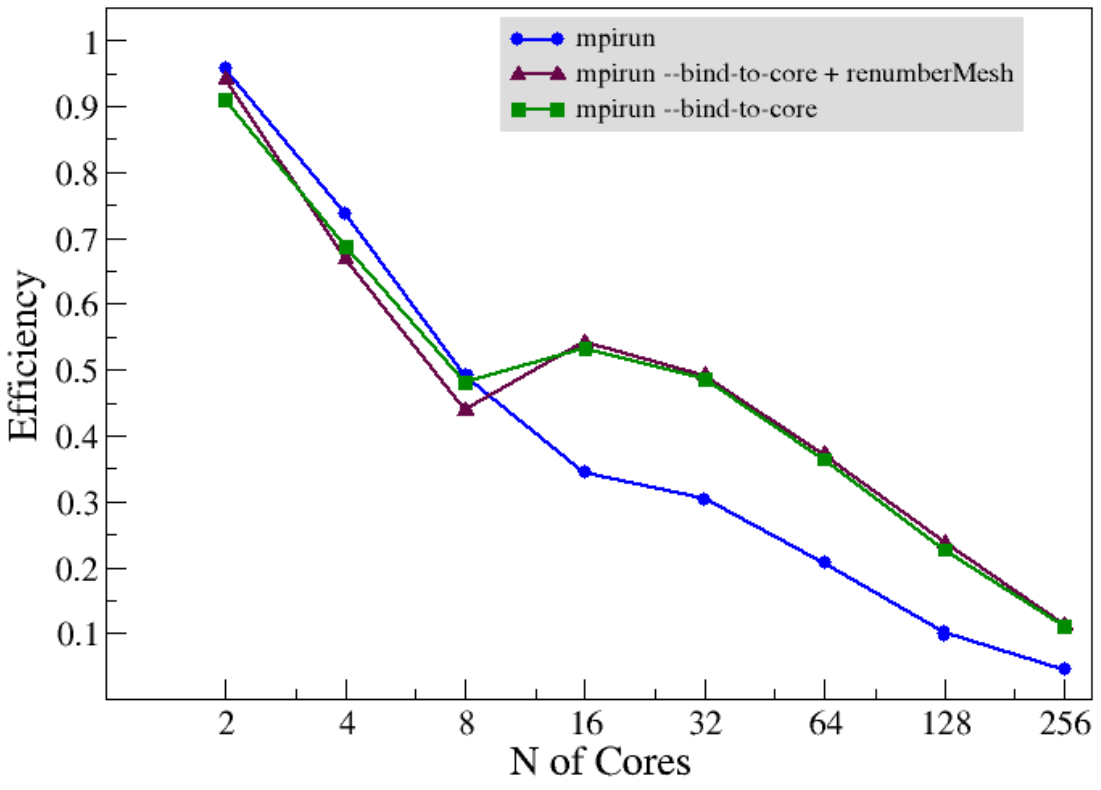


Figure 16. Scaling of rhoPimpleFoam with the low-resolution mesh. The –bind-to-core is an essential MPI option. The renumberMesh OpenFOAM utility can also be used to improve the parallel efficiency.

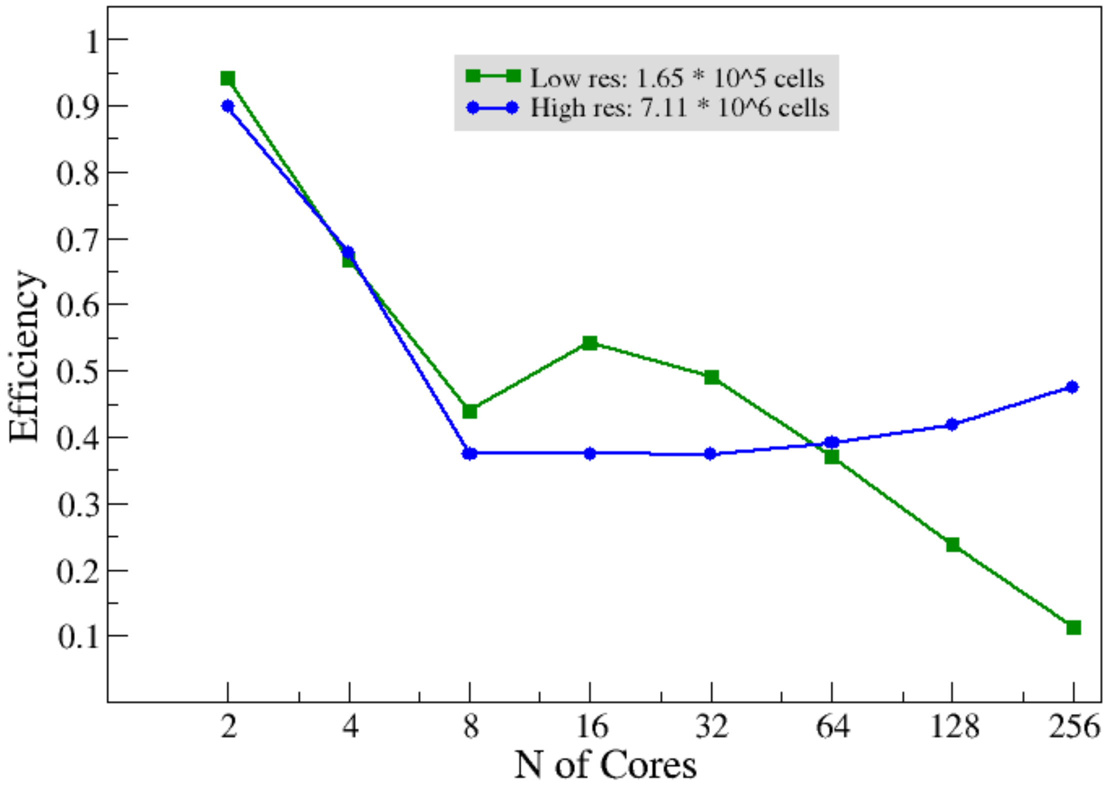


Figure 17. Scaling of rhoPimpleFoam at fixed problem size, increasing the number of processors, for two different resolutions. MPI parallel efficiency is satisfactory for N>8, and 10.000 cells per core; intra-node efficiency is the main concern.

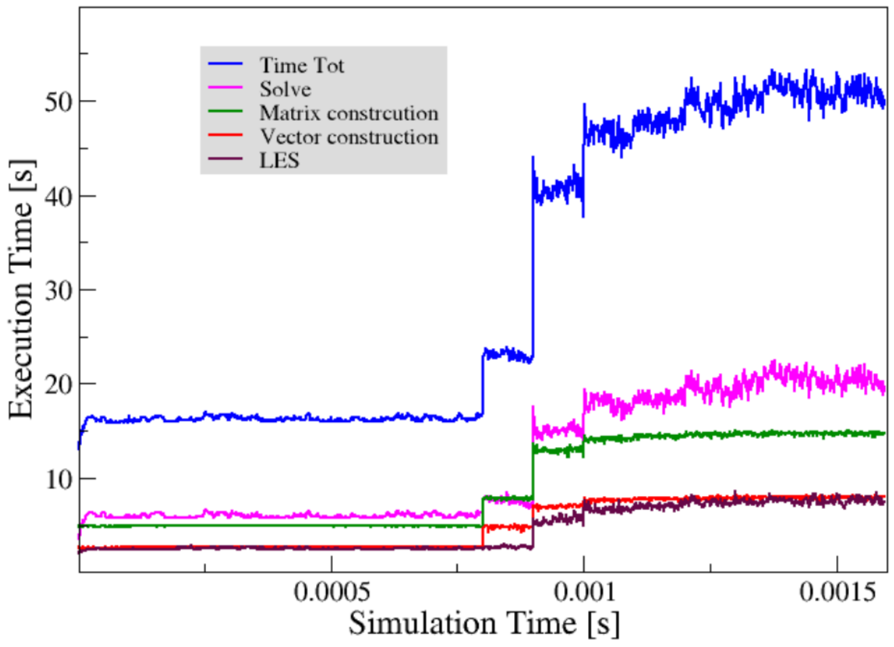


Figure 18. Profiling of an OpenFOAM application on 4 cores, with the high-resolution mesh. Efficiency is about 100% at the beginning of the simulation, when the averaged execution time of every routine is stable. It then starts increasing in a step-wise manner, thus degrading the performance.

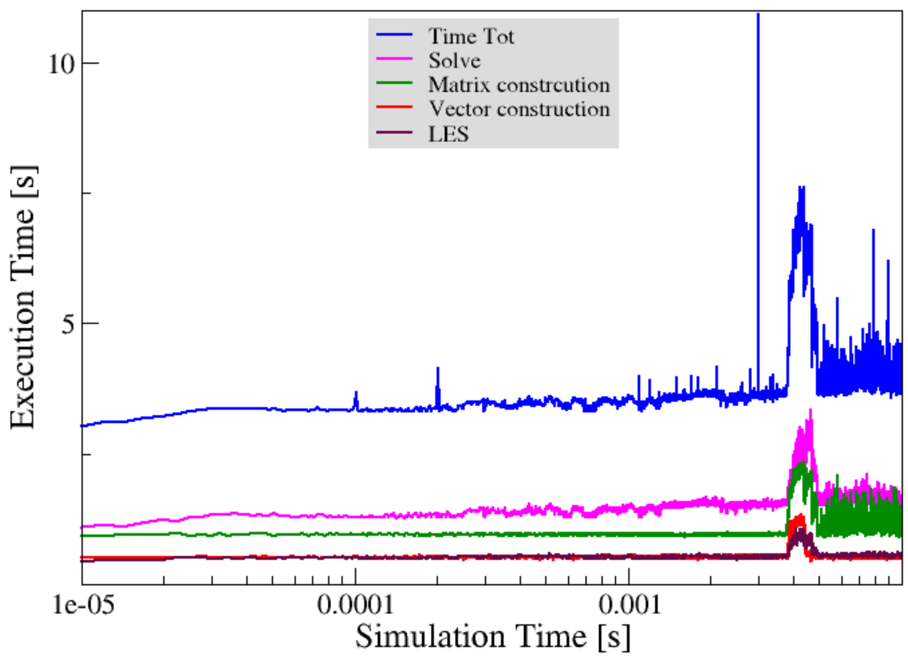


Figure 19. Profiling on 32 cores with the high-resolution mesh. The step-wise structure of Figure 18 has disappeared. The peak at about 0.004 s is probably due to the operating system but has not effect on the global performance.

5.3 ASHEE

As already mentioned the aim of this study is to understand how to run in the most efficient way OpenFOAM and OpenFOAM-like simulations. In particular our interest is focused on ASHEE, a code that numerically simulates the non-equilibrium dynamics of polydisperse gas-particle mixtures forming volcanic plumes [Cerminara et al., 2016]. We repeated for this application the same scaling study of Fig. 17, using two different resolutions at the scale of volcanic plumes, such that the overall number of cells is comparable with the one used in the previous test-case; also the same number of time steps is used to be able to compare results.

Not surprisingly, the outcomes are similar: in Figure 20 we still see the weird behaviour from 1 to 8 cores, i. e. in the intra-node part of the graph. Beyond 8 cores an almost linear scaling (In Figure 20 the horizontal axis has log scale) is retrieved for the high-resolution curve (blue-dotted curve). For the low resolution case (green-dotted line) the performance is worsening for a number of cores greater than 32: again, this means that the problem has become too small to scale properly, a result already found as discussed in the previous section.

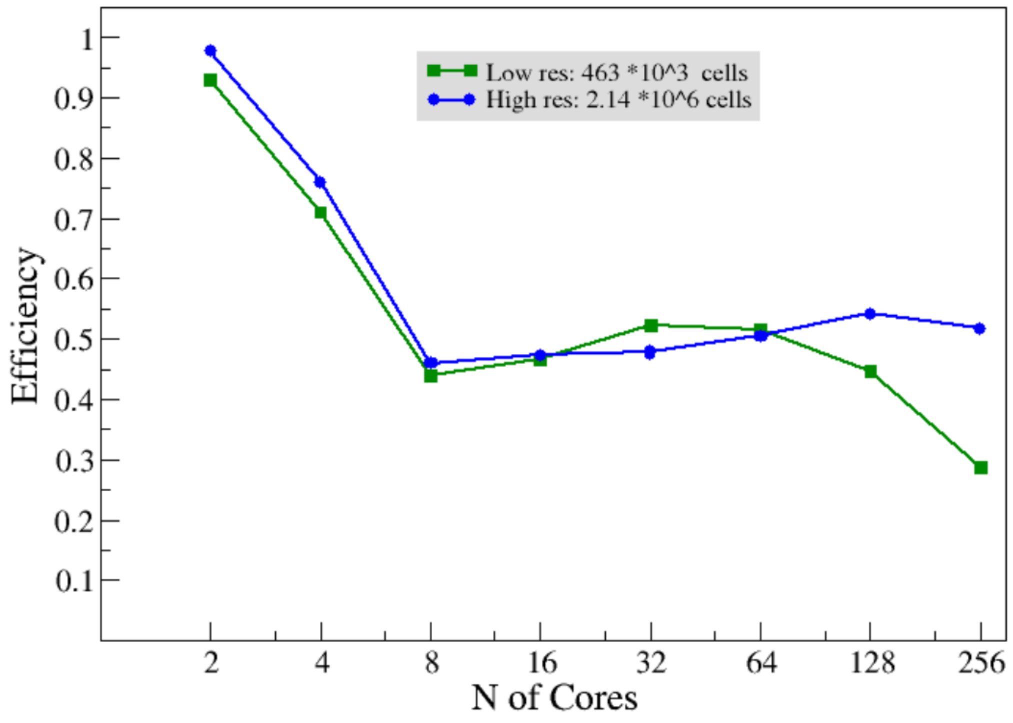


Figure 20. Weak scaling of ASHEE at two different resolutions. Again, the intra-node efficiency drop appears to be the main problem of the Laki architecture.

6. Conclusion

The Laki cluster has been installed in early 2016 at INGV, Sezione di Pisa, to support modelling activity in Computational Fluid Dynamics of volcanic processes. In this report, we have analysed its performance by means of the standard HPL and HPCG standard benchmarks and through one standard OpenFOAM use case and an ASHEE volcanic plume simulation. Results demonstrate that the overall cluster performance is satisfactory, with a peak performance, obtained on the HPL benchmark, of about **4.5 TFLOPS**. On the other hand, scalability properties are not fully satisfactory, showing a significant intra-node performance drop. This is not due to any MPI overhead but it is likely associated to bottlenecks in memory access. To overcome this difficulty, specific parallelization strategies should be implemented to exploit the memory sharing in the node. Overall, the MPI performance is very good, thanks to the low latency and large bandwidth of the InfiniBand interconnection.

Acknowledgements

S.V. Paronuzzi Ticco and M. Cerminara have been supported by OGS (Istituto Nazionale di Oceanografia e Geofisica Sperimentale, Italy) and CINECA under the HPC-TRES (HPC Training and Research for Earth Sciences) program awards n. 2015-03 and 2015-11.

References

Anderson T.E., Culler D.E. and Patterson D.A., (1995). *A case for NOW (networks of workstations*. IEEE Micro, 15(1): 54-64.

Sterling T., Savarese D., Becker D.J., Dorband J.E., Ranawake U.A. and Packer C.V., (1995). *BEOWULF: A parallel workstation for scientific computation.* Proceedings of the 24th International Conference on Parallel Processing, volume I, pages 11-14, Oconomowoc, WI.

Dongarra J., Luszczek P. and Petitet A., (2003). *The LINPACK Benchmark: past, present and future*" Concurrency Computat. Pract. Exper.

Dongarra J. and Heroux M.A. (2013). *Toward a New Metric for Ranking High Performance Computing Systems*. Sandia Report, Unlimited Release.

Cerminara M., Esposti Ongaro T. and Berselli L.C., (2016). *ASHEE-1.0: a compressible, equilibrium-Eulerian model for volcanic ash plumes*. Geosci. Model Dev., 9, 697-730.

A Appendix

The HPCG benchmark is committed to understanding how large systems handle the strain of actual application performance, trying to address the limitations that many centres felt over HPL, i.e. that it did not adequately measure the potential to express real application performance. This might mean that, as a result, one will have less “appealing” numbers to share with the world: this is the reason why only 20 among many supercomputing centres decided to run and share result for this application (Figure 21) back in 2015. Things are changing slowly over time: of the 500 supercomputers that make the Linpack-based Top 500 list in June 2016, the newer HPCG benchmark had over 80 participants. Its becoming more and more evident the relevance of this benchmark, as both vendors and HPC centres began to understand the need to have such results at hand when it comes to forecast the performance expected from real-word applications. To be clearer, in Table 4 results from supercomputers leading the Top500 list are reported. The gap between HPL and HPCG benchmark is striking, and, taking a look at the different architectures beyond the reported numbers is clear that GPU accelerated systems, that tend to perform well on LINPACK, really do not pull the same power on this real-world application-oriented benchmark. As Jack Dongarra, founder of the original HPL Linpack, author and one of the first promoter HPCG, said: “It’s not unlike HPL where GPUs and co-processors have a lower achievable percentage of peak because it’s harder to extract performance from these. It’s not just programming ease either, it’s about the interconnect. When that problems goes away it will change the game dramatically”. Since the rise in accelerators and co-processors (more on that in the context of the Top 500) is not expected to halt soon-meaning these flaws will become far more pronounced in the next years.

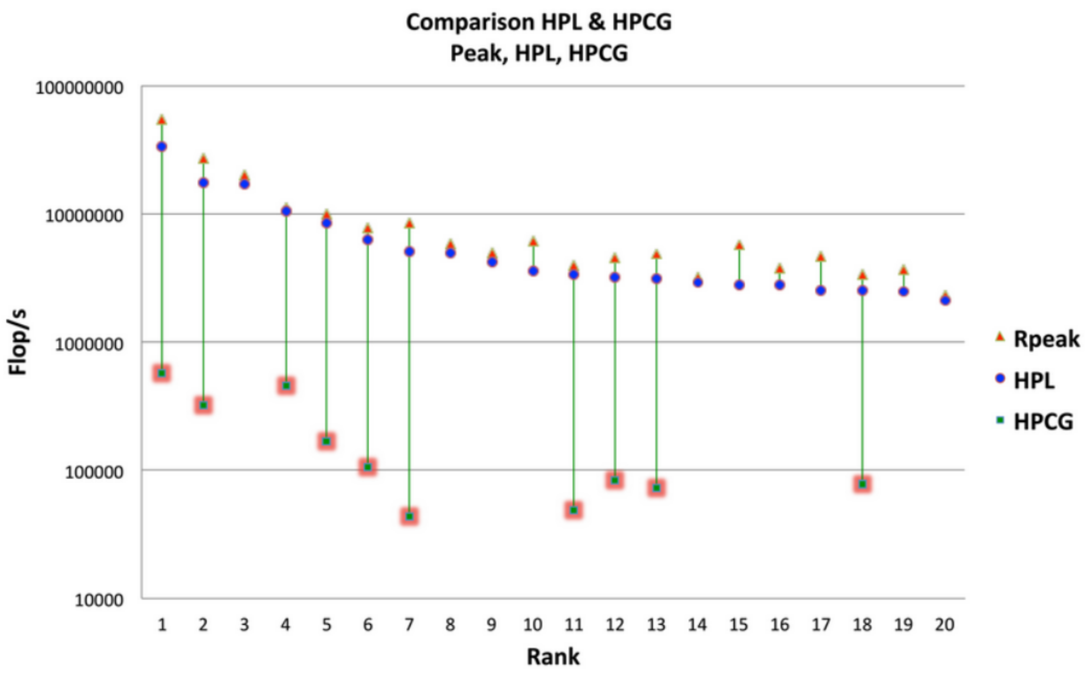


Figure 21. Performance results for the 20 total submissions for November 2014 Top500 supercomputer list. Theoretical peak performance, HPL peak performance, and HPCG peak give rather different numbers.

	Sunway TaihuLight	NUDT Tianhe-2	ORNL Titan	RIKEN K computer
HPL (HPCG) Rank	1 (3)	2 (1)	3 (5)	5 (2)
Theoretical Peak	125 PFLOPS	54.9 PFLOPS	27 PFLOPS	11.2 PFLOPS
HPL	93 PFLOPS	33.86 PFLOPS	17.59 PFLOPS	10.51 PFLOPS
HPCG	0.371 PFLOPS	0.580 PFLOPS	0.322 PFLOPS	0.554 PFLOPS
HPL/HPCG	0.4 %	1.7 %	1.8 %	5.3%
Cores	10.649.600	3.120.000	560.640	705.024
Node	Sunway 260c 1.45 GHz	Intel Xeon 12c 2.2 GHz	Opteron 16c 2.2 Ghz	SPARC64 2 GHz
Accelerators	MPE 4c	Intel Xeon Phi 57c	Nvidia Kepler 14c	*

Table 4. Respective ranking of the first 5 supercomputers in the June 21, 2016 Top500 list. It is evident that the presence of accelerators does not help in real-world computation, better represented by HPCG numbers. Anyway this is not a sign that this kind of architectures is doomed to fail, on the contrary, since a great deal of the interconnect problem with co-processors and GPUs will be a thing of the past, once data never has to leave its chip home.

Coordinamento editoriale e impaginazione
Centro Editoriale Nazionale | INGV

Progetto grafico e redazionale
Daniela Riposati | Laboratorio Grafica e Immagini | INGV

Istituto Nazionale di Geofisica e Vulcanologia